■境界球クラス

```
// BoundingSphere.h
#ifndef BOUNDING_SPHERE_H_
#define BOUNDING_SPHERE_H_

#include <gslib.h>

// 境界球クラス
class BoundingSphere {
public:
    // コンストラクタ
    BoundingSphere() = default;
    // コンストラクタ
    BoundingSphere(float radius, const GSvector3& center = GSvector3{ 0.0f, 0.0f, 0.0f });
    // 平行移動
    BoundingSphere translate(const GSvector3& position) const;
    // 座標変換
    BoundingSphere transform(const GSmatrix4& matrix) const;
    // 重なっているか？
    bool intersects(const BoundingSphere& other) const;
    // デバッグ表示
    void draw() const;

public:
    // 中心座標
    GSvector3   center{ 0.0f, 0.0f, 0.0f };
    // 半径
    float       radius{ 0.0f };
};

#endif


// BoundingSphere.cpp
#include "BoundingSphere.h"

// コンストラクタ
BoundingSphere::BoundingSphere(float radius, const GSvector3& center) :
    radius{ radius }, center{ center } {
}

// 平行移動
BoundingSphere BoundingSphere::translate(const GSvector3& position) const {
    return{ radius, center + position };
}

// 座標変換
BoundingSphere BoundingSphere::transform(const GSmatrix4& matrix) const {
    return{ radius * matrix.getScale().y, matrix.transform(center) };
}

// 交差しているか？
bool BoundingSphere::intersects(const BoundingSphere& other) const {
    return gsCollisionSphereAndSphere(
        &center, radius, &other.center, other.radius) == GS_TRUE;
}

// デバッグ表示
void BoundingSphere::draw() const {
    glPushMatrix();
        glTranslatef(center.x, center.y, center.z);
        glutWireSphere(radius, 16, 16);
    glPopMatrix();
}
```

■アクタークラス

```
// Actor.h
#ifndef ACTOR_H_
#define ACTOR_H_

#include "BoundingSphere.h"
#include <gslib.h>
#include <string>

class IWorld;
enum class EventMessage;

// アクタークラス
class Actor {
public:
    // コンストラクタ
    Actor() = default;
    // 仮想デストラクタ
    virtual ~Actor() {}
    // 更新
    virtual void update(float delta_time);
    // 描画
    virtual void draw() const;
    // 衝突時リアクション
    virtual void react(Actor& other);
    // メッセージ処理
    virtual void handle_message(EventMessage message, void* param);
    // 衝突判定
    void collide(Actor& other);
    // 死亡する
    void die();
    // 衝突しているか？
    bool is_collided(const Actor& other) const;
    // 死亡しているか？
    bool is_dead() const;
    // 名前を取得
    const std::string& name() const;
    // 座標を取得
    GSvector3 position() const;
    // 回転変換行列を取得
    GSmatrix4 rotation() const;
    // 移動量を取得
    GSvector3 velocity() const;
    // 変換行列を取得
    GSmatrix4 pose() const;
    // 衝突判定データを取得
    BoundingSphere body() const;
    // コピー禁止
    Actor(const Actor& other) = delete;
    Actor& operator = (const Actor& other) = delete;

protected:
    // ワールド
    IWorld*         world_{ nullptr };
    // 名前
    std::string     name_{};
    // 座標
    GSvector3       position_{ 0.0f, 0.0f, 0.0f };
    // 回転
    GSmatrix4       rotation_{ GS_MATRIX4_IDENTITY };
    // 移動量
    GSvector3       velocity_{ 0.0f, 0.0f, 0.0f };
    // 衝突判定
    BoundingSphere  body_{};
    // 死亡フラグ
    bool            dead_{ false };
};

#endif
```

```cpp
// Actor.cpp
#include "Actor.h"

// 更新
void Actor::update(float) {}

// 描画
void Actor::draw() const {}

// 衝突時リアクション
void Actor::react(Actor&) {}

// メッセージ処理
void Actor::handle_message(EventMessage, void*) {}

// 衝突判定
void Actor::collide(Actor& other) {
    if (is_collided(other)) {
        react(other);
        other.react(*this);
    }
}

// 死亡する
void Actor::die() {
    dead_ = true;
}

// 衝突判定
bool Actor::is_collided(const Actor& other) const {
    return body().intersects(other.body());
}

// 死亡しているか？
bool Actor::is_dead() const {
    return dead_;
}

// 名前を取得
const std::string& Actor::name() const {
    return name_;
}

// 座標を取得
GSvector3 Actor::position() const {
    return position_;
}

// 回転変換行列を取得
GSmatrix4 Actor::rotation() const {
    return rotation_;
}

// 移動量を取得
GSvector3 Actor::velocity() const {
    return velocity_;
}

// 変換行列を取得
GSmatrix4 Actor::pose() const {
    return rotation().setPosition(position_);
}

// 衝突判定データを取得
BoundingSphere Actor::body() const {
    return body_.transform(pose());
}
```

■フィールドクラス

```
// Field.h
#ifndef FIELD_H_
#define FIELD_H_

#include <gslib.h>

// フィールドクラス
class Field {
public:
    // コンストラクタ
    Field(GSuint bg);
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // フィールド内か？
    bool is_inside(const GSvector3& position) const;
    // フィールド外か？
    bool is_outside(const GSvector3& position) const;

    // コピー禁止
    Field(const Field& other) = delete;
    Field& operator = (const Field& other) = delete;
private:
    // 背景画像
    GSuint  bg_{};
    // スクロール位置
    float   scroll_{ 0.0f };
};

#endif


// Field.cpp
#include "Field.h"
#include "Asset.h"

const float FieldSize{ 150.0f };

// コンストラクタ
Field::Field(GSuint bg) : bg_{ bg }, scroll_{ 0.0f } {
}

// 更新
void Field::update(float delta_time) {
    scroll_ += 0.2f * delta_time;
}

// 描画
void Field::draw() const {
    const GSrect src_rect{ 0.0f, scroll_, 640.0f, 480.0f + scroll_ };
    gsDrawSprite2D(bg_, NULL, &src_rect, NULL, NULL, NULL, 0.0f);
}

// フィールド内か？
bool Field::is_inside(const GSvector3& position) const {
    if (ABS(position.x) >= FieldSize) return false;
    if (ABS(position.y) >= FieldSize) return false;
    return true;
}

// フィールド外か？
bool Field::is_outside(const GSvector3 & position) const {
    return !is_inside(position);
}
```

■フィールドポインタ

```
// FieldPtr.h
#ifndef FIELD_PTR_H_
#define FIELD_PTR_H_

#include <memory>

// フィールドポインタ
class Field;
using FieldPtr = std::shared_ptr<Field>;

// フィールドの作成
template<class T, class... Args>
inline FieldPtr new_field(Args&&... args) {
    return std::make_shared<T>(args...);
}

#endif
```

■カメラクラス

```
// Camera.h
#ifndef CAMERA_H_
#define CAMERA_H_

#include "Actor.h"

// カメラクラス
class Camera : public Actor {
public:
    // コンストラクタ
    explicit Camera(IWorld& world);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
};

#endif
```

```
// Camera.cpp
#include "Camera.h"

// コンストラクタ
Camera::Camera(IWorld& world) {
    world_ = &world;
    name_  = "Camera";
}

// 更新
void Camera::update(float) {
}

// 描画
void Camera::draw() const {
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(
        0.0f, 0.0f, 200.0f,
        0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f
    );
}
```

■ライトクラス

```
// Light.h
#ifndef LIGHT_H_
#define LIGHT_H_

#include "Actor.h"

// ライトクラス
class Light : public Actor {
public:
    // コンストラクタ
    Light(IWorld& world, const GSvector3& position);
    // 描画
    virtual void draw() const override;
};

#endif
```

```
// Light.cpp
#include "Light.h"

// コンストラクタ
Light::Light(IWorld& world, const GSvector3& position) {
    world_ = &world;
    name_  = "Light";
    position_ = position;
}

// 描画
void Light::draw() const {
    static const float ambient[] { 0.2f, 0.2f, 0.2f, 1.0f };
    static const float diffuse[] { 1.0f, 1.0f, 1.0f, 1.0f };
    static const float specular[] { 1.0f, 1.0f, 1.0f, 1.0f };
    float position[] { position_.x, position_.y, position_.z, 0.0f };
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
}
```

■ワールドクラス

```
// World.h
#ifndef WORLD_H_
#define WORLD_H_

#include "IWorld.h"
#include "ActorGroupManager.h"
#include "ActorPtr.h"
#include "FieldPtr.h"
#include <functional>

enum class EventMessage;

// ワールドクラス
class World : public IWorld {
public:
    // イベントメッセージリスナー
    using EventMessageListener = std::function<void(EventMessage, void*)>;
public:
    // コンストラクタ
    World();
    // 初期化
    void initialize();
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // メッセージ処理
    void handle_message(EventMessage message, void* param);
    // メッセージリスナーの登録
    void add_event_message_listener(EventMessageListener listener);
    // フィールドの追加
    void add_field(const FieldPtr& field);
    // カメラの追加
    void add_camera(const ActorPtr& camera);
    // ライトの追加
    void add_light(const ActorPtr& light);
    // 消去
    void clear();

    // アクターの追加
    virtual void add_actor(ActorGroup group, const ActorPtr& actor) override;
    // アクターの検索
    virtual ActorPtr find_actor(ActorGroup group, const std::string& name) const override;
    // アクター数の取得
    virtual unsigned int count_actor(ActorGroup group) const override;
    // メッセージの送信
    virtual void send_message(EventMessage message, void* param = nullptr) override;
    // フィールドを取得
    virtual Field& field() override;

    // コピー禁止
    World(const World& other) = delete;
    World& operator = (const World& other) = delete;
private:
    // アクターマネージャー
    ActorGroupManager   actors_;
    // ライト
    ActorPtr            light_;
    // カメラ
    ActorPtr            camera_;
    // フィールド
    FieldPtr            field_;
    // メッセージリスナ
    EventMessageListener listener_ { [](EventMessage, void*) {} };
};

#endif
```

```cpp
// World.cpp
#include "World.h"
#include "Field.h"
#include "ActorGroup.h"
#include "Actor.h"

// コンストラクタ
World::World() {
    initialize();
}

// 初期化
void World::initialize() {
    clear();
    actors_.add(ActorGroup::Player);
    actors_.add(ActorGroup::PlayerBullet);
    actors_.add(ActorGroup::Enemy);
    actors_.add(ActorGroup::EnemyBullet);
    actors_.add(ActorGroup::Effect);
}

// 更新
void World::update(float delta_time) {
    field_->update(delta_time);
    actors_.update(delta_time);
    actors_.collide(ActorGroup::Player, ActorGroup::Enemy);
    actors_.collide(ActorGroup::Player, ActorGroup::EnemyBullet);
    actors_.collide(ActorGroup::PlayerBullet, ActorGroup::Enemy);
    actors_.remove();
    camera_->update(delta_time);
    light_->update(delta_time);
}

// 描画
void World::draw() const {
    camera_->draw();
    light_->draw();
    field_->draw();
    actors_.draw();
}

// メッセージ処理
void World::handle_message(EventMessage message, void* param) {
    // ワールドのメッセージ処理
    listener_(message, param);
    // アクターのメッセージ処理
    actors_.handle_message(message, param);
    camera_->handle_message(message, param);
    light_->handle_message(message, param);
}

// イベントメッセージリスナーの登録
void World::add_event_message_listener(EventMessageListener listener) {
    listener_ = listener;
}

// フィールドの追加
void World::add_field(const FieldPtr& field) {
    field_ = field;
}

// カメラの追加
void World::add_camera(const ActorPtr& camera) {
    camera_ = camera;
}

// ライトの追加
void World::add_light(const ActorPtr& light) {
    light_ = light;
}
```

```
// 消去
void World::clear() {
    actors_.clear();
    field_ = nullptr;
    light_ = nullptr;
    camera_ = nullptr;
    listener_ = [](EventMessage, void*) {};
}

// アクターの追加
void World::add_actor(ActorGroup group, const ActorPtr& actor) {
    actors_.add(group, actor);
}

// アクターの検索
ActorPtr World::find_actor(ActorGroup group, const std::string & name) const {
    return actors_.find(group, name);
}

// アクター数を返す
unsigned int World::count_actor(ActorGroup group) const {
    return actors_.count(group);
}

// メッセージの送信
void World::send_message(EventMessage message, void* param) {
    handle_message(message, param);
}

// フィールドの取得
Field& World::field() {
    return *field_;
}
```

■隕石クラス

```
// Asteroid.h
#ifndef ASTEROID_H_
#define ASTEROID_H_

#include "Actor.h"

// 隕石クラス
class Asteroid : public Actor {
public:
    // コンストラクタ
    Asteroid(IWorld& world, const GSvector3& position, const GSvector3& velocity);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
    // 衝突リアクション
    virtual void react(Actor& other) override;
private:
    // 角度
    float       angle_{ 0.0f };
};

#endif


// Asteroid.cpp
#include "Asteroid.h"
#include "IWorld.h"
#include "ActorGroup.h"
#include "Asset.h"
#include "IWorld.h"
#include "Field.h"
#include "Explosion.h"
// コンストラクタ
Asteroid::Asteroid(IWorld& world, const GSvector3& position, const GSvector3& velocity) {
    world_    = &world;
    name_     = "Asteroid";
    position_ = position;
    velocity_ = velocity;
    body_     = BoundingSphere{ 8.0f };
    angle_    = 0.0f;
}
// 更新
void Asteroid::update(float delta_time) {
    position_ += velocity_ * delta_time;
    angle_ += delta_time;
    // 画面外であれば死亡
    if (world_->field().is_outside(position_)) {
        die();
    }
}
// 描画
void Asteroid::draw() const {
    glPushMatrix();
        glMultMatrixf(pose());
        glRotatef(angle_ * 0.5f, 0.0f, 0.0f, 1.0f);
        glRotatef(angle_ * 2.0f, 1.0f, 0.0f, 0.0f);
        gsDrawMesh(Mesh_Asreroid01);
    glPopMatrix();
}
// 衝突リアクション
void Asteroid::react(Actor&) {
    gsPlaySE(Se_ExplosionAsteroid);
    world_->add_actor(ActorGroup::Effect, new_actor<Explosion>(*world_, position_));
    die();
}
```

■プレーヤークラス

```
// Player.h
#ifndef PLAYER_H_
#define PLAYER_H_

#include "Actor.h"

// プレーヤ
class Player : public Actor {
public:
    // コンストラクタ
    Player(IWorld& world, const GSvector3& position);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
    // 衝突処理
    virtual void react(Actor& other) override;
};

#endif

// Player.cpp
#include "Player.h"
#include "PlayerBullet.h"
#include "IWorld.h"
#include "ActorGroup.h"
#include "Explosion.h"
#include "Asset.h"

// 移動範囲
const float MovingRangeX{ 100.f };
const float MovingRangeY{ 70.f };

// コンストラクタ
Player::Player(IWorld& world, const GSvector3 & position) {
    world_    = &world;
    name_     = "Player";
    position_ = position;
    body_     = BoundingSphere{ 5.0f };
}

// 更新
void Player::update(float delta_time) {
    GSvector3 velocity{ 0.0f, 0.0f, 0.0f };
    if (gsGetKeyState(GKEY_LEFT) == GS_TRUE) {
        velocity.x = -1.0f;
    }
    if (gsGetKeyState(GKEY_RIGHT) == GS_TRUE) {
        velocity.x = 1.0f;
    }
    if (gsGetKeyState(GKEY_UP) == GS_TRUE) {
        velocity.y = 1.0f;
    }
    if (gsGetKeyState(GKEY_DOWN) == GS_TRUE) {
        velocity.y = -1.0f;
    }
    position_ += velocity.getNormalized() * delta_time;
    position_.x = CLAMP(position_.x, -MovingRangeX, MovingRangeX);
    position_.y = CLAMP(position_.y, -MovingRangeY, MovingRangeY);
    if (gsGetKeyTrigger(GKEY_Z) == GS_TRUE) {
        gsPlaySE(Se_WeaponPlayer);
        world_->add_actor(
            ActorGroup::PlayerBullet,
            new_actor<PlayerBullet>(*world_, position_, GSvector3(0.0f, 4.0f, 0.0f)));
    }
}
```

```
// 描画
void Player::draw() const {
    glPushMatrix();
        glMultMatrixf(pose());
        glRotatef(180.0f, 0.0f, 0.0f, 1.0f);
        glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
        gsDrawMesh(0);
    glPopMatrix();
}

// 衝突処理
void Player::react(Actor &) {
    gsPlaySE(Se_ExplosionPlayer);
    world_->add_actor(ActorGroup::Effect, new_actor<Explosion>(*world_, position_));
    die();
}
```

■敵クラス

```
// Enemy.h
#ifndef ENEMY_H_
#define ENEMY_H_

#include "Actor.h"

// 敵
class Enemy : public Actor {
public:
    // コンストラクタ
    Enemy(IWorld& world, const GSvector3&  position);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
    // 衝突リアクション
    virtual void react(Actor& other) override;

private:
    // 移動中タイマ
    float       move_timer_{ 0.0f };
    // 弾発射タイマ
    float       bullet_timer_{ 0.0f };
};

#endif
```

```cpp
// Enemy.cpp
#include "Enemy.h"
#include "EnemyBullet.h"
#include "IWorld.h"
#include "ActorGroup.h"
#include "Asset.h"
#include "Explosion.h"
#include "Field.h"

// コンストラクタ
Enemy::Enemy(IWorld& world, const GSvector3 & position) {
    world_       = &world;
    name_        = "Enemy";
    position_    = position;
    velocity_    = GSvector3{ 0.0f, -1.0f, 0.0f };
    body_        = BoundingSphere{ 5.0f };
    move_timer_   = gsRandf(0.0f, 60.0f);
    bullet_timer_ = gsRandf(0.0f, 60.0f);
}

// 更新
void Enemy::update(float delta_time) {
    // ランダムな間隔でプレーヤに向かう
    if (move_timer_ <= 0.0f) {
        const auto& player = world_->find_actor(ActorGroup::Player, "Player");
        if (player != nullptr) {
            const auto& to_player = (player->position() - position_).getNormalized();
            velocity_ = GSvector3(to_player.x, -2.0f, 0.0f).getNormalized();
        }
        move_timer_ = gsRandf(30.0f, 120.0f);
    }
    move_timer_ -= delta_time;

    // ランダムな間隔で弾を発射
    if (bullet_timer_ <= 0.0f) {
        gsPlaySE(Se_WeaponEnemy);
        world_->add_actor(
            ActorGroup::EnemyBullet,
            new_actor<EnemyBullet>(*world_, position_, GSvector3(0.0f, -4.0f, 0.0f)));
        bullet_timer_ = gsRand(20.0f, 60.0f);
    }
    bullet_timer_ -= delta_time;

    // 移動する
    position_ += velocity_ * delta_time;

    // 画面外であれば死亡
    if (world_->field().is_outside(position_)) {
        die();
    }
}

// 描画
void Enemy::draw() const {
    glPushMatrix();
        glMultMatrixf(pose());
        glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
        gsDrawMesh(Mesh_EnemyO1);
    glPopMatrix();
}

// 衝突リアクション
void Enemy::react(Actor &) {
    gsPlaySE(Se_ExplosionEnemy);
    world_->add_actor(ActorGroup::Effect, new_actor<Explosion>(*world_, position_));
    die();
}
```

■プレーヤ弾クラス

```
// PlyerBullet.h
#ifndef PLAYER_BULLET_H_
#define PLAYER_BULLET_H_

#include "Actor.h"

// 弾クラス
class PlayerBullet : public Actor {
public:
    // コンストラクタ
    PlayerBullet(IWorld& world, const GSvector3& position, const GSvector3& velocity);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
    // 衝突リアクション
    virtual void react(Actor& other) override;
};

#endif


// PlyerBullet.cpp
#include "PlayerBullet.h"
#include "IWorld.h"
#include "Field.h"
#include "Asset.h"

// コンストラクタ
PlayerBullet::PlayerBullet(IWorld& world,  const GSvector3& position, const GSvector3& velocity) {
    world_ = &world;
    name_ = "PlayerBullet";
    position_ = position;
    velocity_ = velocity;
    body_ = BoundingSphere{ 1.0f };
}

// 更新
void PlayerBullet::update(float delta_time) {
    position_ += velocity_ * delta_time;
    if (world_->field().is_outside(position_)) {
        die();
    }
}

// 描画
void PlayerBullet::draw() const {
    glPushAttrib(GL_ENABLE_BIT | GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_LIGHTING);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glEnable(GL_BLEND);
    static const GSrect spriteRect{-5.0f, 5.0f, 5.0f, -5.0f};
    gsDrawSprite3D(Texture_EffectLazerOrange, &position_, &spriteRect, NULL, NULL, NULL, 0.0f);
    glPopAttrib();
}

// 衝突処理
void PlayerBullet::react(Actor &) {
    die();
}
```

■敵弾クラス

```
// EnemyBullet.h
#ifndef ENEMY_BULLET_H_
#define ENEMY_BULLET_H_

#include "Actor.h"

// 弾クラス
class EnemyBullet : public Actor {
public:
    // コンストラクタ
    EnemyBullet(IWorld& world, const GSvector3& position, const GSvector3& velocity);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
    // 衝突リアクション
    virtual void react(Actor & other) override;
};

#endif


// EnemyBullet.cpp
#include "EnemyBullet.h"
#include "Asset.h"
#include "IWorld.h"
#include "Field.h"

// コンストラクタ
EnemyBullet::EnemyBullet(IWorld& world, const GSvector3& position, const GSvector3& velocity) {
    world_    = &world;
    name_     = "EnemyBullet";
    position_ = position;
    velocity_ = velocity;
    body_     = BoundingSphere{ 1.0f };
}

// 更新
void EnemyBullet::update(float delta_time) {
    position_ += velocity_ * delta_time;
    // 画面外であれば死亡
    if (world_->field().is_outside(position_)) {
        die();
    }
}

// 描画
void EnemyBullet::draw() const {
    glPushAttrib(GL_ENABLE_BIT | GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_LIGHTING);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glEnable(GL_BLEND);
    static const GSrect spriteRect{-5.0f, 5.0f, 5.0f, -5.0f};
    gsDrawSprite3D(Texture_EffectLazerCyan, &position_, &spriteRect, NULL, NULL, NULL, 0.0f);
    glPopAttrib();
}

// 衝突処理
void EnemyBullet::react(Actor &) {
    die();
}
```

■爆発炎クラス

```
// ExplosionPartFire.h
#ifndef EXPLOSION_PART_FIRE_H_
#define EXPLOSION_PART_FIRE_H_

#include "Actor.h"

// 爆発炎クラス
class ExplosionPartFire : public Actor {
public:
    // コンストラクタ
    ExplosionPartFire(IWorld& world, const GSvector3& position, const GSvector3& velocity);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
private:
    // 消滅タイマ
    float       timer_{ 0.0f };
};

#endif


// ExplosionPartFire.cpp
#include "ExplosionPartFire.h"
#include "Asset.h"
#include <GSeasing.h>

const float MoveTime{ 16.0f };

// コンストラクタ
ExplosionPartFire::ExplosionPartFire(IWorld& world, const GSvector3& position, const GSvector3& velocity) {
    world_    = &world;
    name_ = "ExplosionPartFire";
    position_ = position;
    velocity_ = velocity;
}

// 更新
void ExplosionPartFire::update(float delta_time) {
    position_ += velocity_ * delta_time;
    if (timer_ > MoveTime) {
        die();
    }
    timer_ += delta_time;
}

// 描画
void ExplosionPartFire::draw() const {
    glPushAttrib(GL_ENABLE_BIT | GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_LIGHTING);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glEnable(GL_BLEND);
    float t = timer_ / MoveTime;
    static const GSrect body{ -5.0f, 0.0f, 5.0f, -10.0f };
    static const GScolor start_color{ 1.0f, 1.0f, 1.0f, 1.0f };
    static const GScolor end_color{ 1.0f, 0.5f, 0.5f, 0.0f };
    GScolor color = start_color.lerp(end_color, gsEasingInExpo(t));
    static const GSvector2 start_scale{ 1.0f, 1.0f };
    static const GSvector2 end_scale{ 1.0f, 1.5f };
    GSvector2 scale = start_scale.lerp(end_scale, t);
    float angle = GSvector2(velocity_.x, velocity_.y).getDirection() - 90.0f;
    gsDrawSprite3D(Texture_EffectLazerOrange, &position_, &body, NULL, &color, &scale, angle);
    glPopAttrib();
}
```

■爆発フラッシュクラス

```
// ExplosionPartFlash.h
#ifndef EXPLOSION_PART_FLASH_H_
#define EXPLOSION_PART_FLASH_H_

#include "Actor.h"

// 爆発フラッシュクラス
class ExplosionPartFlash : public Actor {
public:
    // コンストラクタ
    ExplosionPartFlash(IWorld& world, const GSvector3& position);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
private:
    // 消滅タイマ
    float       timer_{ 0.0f };
};

#endif


// ExplosionPartFlash.cpp
#include "ExplosionPartFlash.h"
#include "Asset.h"
#include <GSeasing.h>

const float FlashTime{ 20.0f };

// コンストラクタ
ExplosionPartFlash::ExplosionPartFlash(IWorld& world, const GSvector3& position) {
    world_    = &world;
    name_     = "ExplosionPartFlash";
    position_ = position;
}

// 更新
void ExplosionPartFlash::update(float delta_time) {
    if (timer_ > FlashTime) {
        die();
    }
    timer_ += delta_time;
}

// 描画
void ExplosionPartFlash::draw() const {
    glPushAttrib(GL_ENABLE_BIT | GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_LIGHTING);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glEnable(GL_BLEND);
    float t = timer_ / FlashTime;
    static const GSrect body{ -20.0f, 20.0f, 20.0f, -20.0f };
    static const GScolor start_color{ 1.0f, 1.0f, 1.0f, 1.0f };
    static const GScolor end_color{ 0.8f, 0.0f, 0.0f, 0.0f };
    GScolor color = start_color.lerp(end_color, gsEasingInOutCubic(t));
    static const GSvector2 start_scale{ 0.5f, 0.5f };
    static const GSvector2 end_scale{ 1.0f, 1.0f };
    GSvector2 scale = start_scale.lerp(end_scale, gsEasingOutBack(t));
    gsDrawSprite3D(Texture_EffectFlash, &position_, &body, NULL, &color, &scale, 0.0f);
    glPopAttrib();
}
```

■爆発エフェクトクラス

```
// Explosion.h
#ifndef EXPLOSION_H_
#define EXPLOSION_H_

#include "Actor.h"

class Explosion : public Actor {
public:
    // コンストラクタ
    Explosion(IWorld& world, const GSvector3& position);
    // 更新
    virtual void update(float delta_time) override;
};

#endif


// Explosion.cpp
#include "Explosion.h"
#include "IWorld.h"
#include "ActorGroup.h"
#include "ExplosionPartFlash.h"
#include "ExplosionPartFire.h"
#include "Asset.h"

// コンストラクタ
Explosion::Explosion(IWorld& world, const GSvector3& position) {
    world_    = &world;
    name_     = "Explosion";
    position_ = position;
}

// 更新
void Explosion::update(float delta_time) {
    for (float pitch = 0.0f; pitch < 360.0f; pitch += 20.0f) {
        world_->add_actor(ActorGroup::Effect, new_actor<ExplosionPartFlash>(*world_, position_));
        const auto& velocity = GSvector3::createFromPitchYaw(pitch + gsRandf(-10.0f, 10.0f), 90.0f) * gsRandf(3.0f, 4.0f);
        world_->add_actor(ActorGroup::Effect, new_actor<ExplosionPartFire>(*world_, position_, velocity));
    }
    die();
}
```

■CSVリーダークラス

```
// CSVReader.h
#ifndef CSV_READER_H_
#define CSV_READER_H_

#include <vector>
#include <string>

// CSVリーダークラス
class CsvReader {
public:
    // コンストラクタ
    CsvReader()= default;
    // コンストラクタ
    CsvReader(const std::string& file_name);
    // ファイルの読み込み
    void load(const std::string& file_name);
    // データの取得
    const std::string& get(int row, int column) const;
    // データの取得
    int geti(int row, int column) const;
    // データの取得
    float getf(int row, int column) const;
    // 行数を返す
    int rows() const;
    // 列数を返す
    int columns(int row = 0) const;

private:
    using Row = std::vector<std::string>;
    using Rows = std::vector<Row>;
    Rows rows_;
};

#endif
```

```cpp
// CSVReader.cpp
#include "CSVReader.h"
#include <fstream>
#include <sstream>
#include <stdexcept>

// コンストラクタ
CsvReader::CsvReader(const std::string& file_name) {
    load(file_name);
}

// ファイルの読み込み
void CsvReader::load(const std::string& file_name) {
    std::ifstream file(file_name);
    if (!file) throw std::runtime_error("CSVファイルがオープンできませんでした");
    rows_.clear();
    std::string line;
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string value;
        Row row;
        while (std::getline(ss, value, ',')) {
            row.push_back(value);
        }
        rows_.push_back(row);
    }
}

// データの取得
const std::string& CsvReader::get(int row, int column) const {
    return rows_[row][column];
}

// データの取得
int CsvReader::geti(int row, int column) const {
    return std::stoi(get(row, column));
}

// データの取得
float CsvReader::getf(int row, int column) const {
    return std::stof(get(row, column));
}

// 行数を返す
int CsvReader::rows() const {
    return (int)rows_.size();
}

// 列数を返す
int CsvReader::columns(int row) const {
    return (int)rows_[row].size();
}
```

■敵生成クラス

```
// EnemyGenerator.h
#ifndef ENEMY_GENERATOR_H_
#define ENEMY_GENERATOR_H_

#include "Actor.h"
#include "CSVReader.h"

// 敵生成クラス
class EnemyGenerator : public Actor {
public:
    // コンストラクタ
    EnemyGenerator(IWorld& world, const std::string& fileName);
    // 更新
    virtual void update(float delta_time) override;

private:
    // 敵の生成
    void generate();

private:
    // 出現データ
    CsvReader   csv_;
    // 現在の読み込み位置
    int         current_row_{ 0 };
    // 出現タイマ
    float       timer_{ 0.0f };
};

#endif
```

```cpp
// EnemyGenerator.cpp
#include "EnemyGenerator.h"
#include "ActorGroup.h"
#include "IWorld.h"
#include "Enemy.h"
#include "Asteroid.h"

const int CsvTime{ 0 };
const int CsvName{ 1 };
const int CsvPosition{ 2 };
const int CsvVelocity{ 5 };

// コンストラクタ
EnemyGenerator::EnemyGenerator(IWorld& world, const std::string & fileName) {
    world_ = &world;
    name_  = "EnemeyGenerator";
    csv_.load(fileName);
}

// 更新
void EnemyGenerator::update(float delta_time) {
    while (current_row_ < csv_.rows() && (csv_.getf(current_row_, CsvTime) * 60.0f) <= timer_) {
        generate();
        ++current_row_;
    }
    timer_ += delta_time;
}

// 敵の生成
void EnemyGenerator::generate() {
    const std::string& name = csv_.get(current_row_, CsvName);
    const GSvector3 position{
        csv_.getf(current_row_, CsvPosition + 0),
        csv_.getf(current_row_, CsvPosition + 1),
        csv_.getf(current_row_, CsvPosition + 2) };

    if (name == "Enemy") {
        world_->add_actor(ActorGroup::Enemy,
            new_actor<Enemy>(*world_, position));
    } else if (name == "Asteroid") {
        const GSvector3 velocity{
            csv_.getf(current_row_, CsvVelocity + 0),
            csv_.getf(current_row_, CsvVelocity + 1),
            csv_.getf(current_row_, CsvVelocity + 2) };
        world_->add_actor(ActorGroup::Enemy,
            new_actor<Asteroid>(*world_, position, velocity));
    }
}
```

■アセット用ヘッダファイル

```
#ifndef ASSET_H_
#define ASSET_H_

// メッシュデータ
enum {
    Mesh_Player,
    Mesh_Enemy01,
    Mesh_Enemy02,
    Mesh_Asreroid01,
    Mesh_Asreroid02,
    Mesh_Asreroid03
};

// テクスチャデータ
enum {
    // 2Dスプライト
    Texture_BgTileNebulaGreen,

    // 3Dスプライト
    Texture_EffectLazerCyan,
    Texture_EffectLazerOrange,
    Texture_EffectJetCore,
    Texture_EffectJetFlare,
    Texture_EffectFlash,
    Texture_EffectShockwave,
    Texture_EffectSparkLarge,
    Texture_EffectSparkSmall,
    Texture_EffectSter,
    Texture_EffectEnginePulse
};

// ＢＧＭデータ
enum {
    Music_BackGround
};

// ＳＥデータ
enum {
    Se_ExplosionPlayer,
    Se_ExplosionEnemy,
    Se_ExplosionAsteroid,
    Se_WeaponEnemy,
    Se_WeaponPlayer
};

#endif
```